CS388G Problem Set #1

Nidhi Kadkol (nk9368)

February 9, 2019

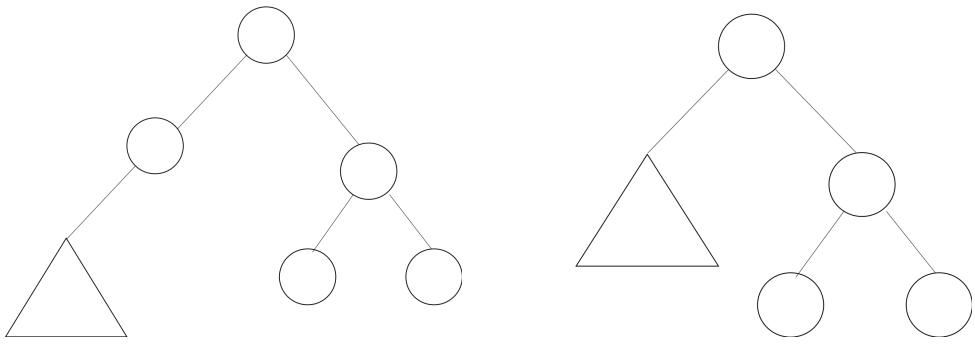
1(a)

In a binary tree every node is either a leaf (has 0 children) or an internal node (has 1 or 2 children).

We define a ***full binary tree*** as a tree in which every node is either a leaf or has 2 children, and a ***complete binary tree*** as a full binary tree in which every level except possibly the last level is filled.

We prove the statement by showing that an ℓ-leaf binary tree has the smallest EPL (External Path Length) when it is a complete binary tree, and that the EPL of a complete binary tree is $O(\ell \lg \ell)$.

Claim 1. An ℓ-leaf binary tree has the smallest EPL when it is a complete binary tree.



(a) Tree 1 (b) Tree 2

Figure 1: We can reduce the External Path Length of a non-full binary tree by replacing the nodes which have only 1 child with their respective children.

Proof. We first prove by contradiction that a full binary tree has a smaller EPL than a non-full binary tree. Suppose it is possible that there exists an ℓ-leaf binary tree which is not a full tree, and has the smallest possible EPL among all non-full binary trees. Since it is not a full tree, there exists a node which has only 1 child. If we replace this node with its child as shown in figure 1, we see that the depths of the leaves in the node's sub-tree get

reduced by 1. Furthermore, the number of leaves in the tree remains the same. Hence, we get an ℓ-leaf tree with a smaller EPL than our original tree thus contradicting our initial assumption. Hence, for a given number of ℓ leaves, a full binary tree has a smaller EPL than any non-full binary tree.

We now show that for ℓ leaves, to build a tree with minimum EPL we have to build a complete binary tree.

Consider $\ell = 1$. A tree with minimum EPL would be a single node. This is a complete tree. Now consider an l-leaf complete tree. We want to add another leaf to this tree. To do this, we split an existing leaf node into 2 children. We want to minimize the EPL, so we split the leaf at the level with smallest depth. As we add more leaves we end up filling up a level completely before moving on to the next level, thus building a complete binary tree. If we use this minimizing EPL strategy to add leaves starting from the beginning of the tree's formation, we maintain a complete binary tree for every leaf we add. □

Claim 2. In a complete binary tree, the depth is $O(\lg \ell)$.

Proof. Every level of a complete binary tree is filled up before we begin adding nodes to the next level. So the number of nodes at each level is twice the number of nodes at the previous level. The number of nodes
at level $0 = 1 = 2^0$,
at level $1 = 2{*}1 = 2^1$,
at level $2 = 2{*}2 = 2^2$,
at level $3 = 2 * 2^2 = 2^3$,
...
at level i $= 2 * 2^{i-1} = 2^i$
...
at level $d - 1 = 2 * 2^{d-2} = 2^{d-1}$
where d is the depth of the tree (the last level of the tree). At level $d - 1$, at least one node has 2 children (or this level would be the last level) and all the other nodes in this level are either leaves or have 2 children. Let the number of nodes at level $d - 1$ that have children be k. Then the number of leaves in the tree = (Number of leaves in level $d - 1$) + (Number of leaves in level d). Each parent node in level $d - 1$ will have 2 children, so the number of leaves in level $d - 1$ will be $2k$.

$$
\begin{aligned}
& \ell = 2^{d-1} - k + 2k \\
\implies & \ell = 2^{d-1} + k \\
\implies & \ell > 2^{d-1} \\
\implies & \lg \ell > d - 1 \\
\implies & d < \lg \ell + 1 \\
& \leq c \lg \ell \qquad \text{for } l \geq 2 \text{ and } c = 2 \\
\implies & d = O(\lg \ell)
\end{aligned}
$$

□

Since every leaf of a complete binary tree is either at depth d or at depth $d - 1$ and $d = O(\lg \ell)$, thus the depth of any leaf in a complete binary tree is $O(\lg \ell)$.

Since the External Path Length is defined as the sum of the depths of all the leaves, it follows that the External Path Length of a complete binary tree is $O(\ell \lg \ell)$. We have proved that among all possible trees, a complete binary tree has the smallest External Path Length for ℓ leaves. Thus, for any ℓ-leaf binary tree, the EPL is greater than or equal to the result we derived for a complete binary tree. Hence, any ℓ-leaf binary tree has external path length $\Omega(\ell \lg \ell)$.

1(b)

Consider an n-sized input to a comparison-based sorting algorithm. We represent the steps of this algorithms by a binary tree, where each branch denotes a comparison, and each node denotes the set of permutations of the input that are consistent with the outcomes of the comparisons made at every branch until that node. Thus, a node at depth i contains the set of permutations that are valid after i comparisons are made.

Thus, such a tree will have $n!$ leaves, and the path from the root to a leaf denotes the comparisons made to reach that particular sorted output.

The average number of steps taking for a comparison-based sorting algorithm is the sum of the number of steps taken to reach each possible output by the number of possible outputs. Relating this to our tree, it is the external path length divided by the number of leaves.

$$
\begin{aligned}
&\text{Number of leaves} = n! \\
\therefore\ &\text{External path length} &&= \Omega(n! \lg(n!)) \\
&&&= \Omega(n! \, n \lg n) \text{ (Stirling's approximation)}
\end{aligned}
$$

For any comparison-based sorting algorithm,

$$
\begin{aligned}
\text{Average-case complexity} &= \frac{\text{(Sum of comparisons for each permutation)}}{\text{(Number of possible permutations)}} \\
&= \frac{\text{(External path length)}}{\text{(Number of leaves)}} \\
&= \frac{\Omega(n! \, n \lg n)}{n!} \\
&= \Omega(n \lg n)
\end{aligned}
$$

2(a)

Proof. Solving for n_0, we get $n_0 = 5$. Thus, our recurrence can be written as:

$$T(n) = \begin{cases} n^2 & \text{if } 1 \leq n \leq 5 \\ 2T\left(\left\lfloor \frac{n}{2} + \sqrt{n} \right\rfloor\right) + n & \text{if } n \geq 5 \end{cases}$$

To prove that $T(n) = O(n\log n)$, we need to show that there exist positive constants c and m_0 such that

$$T(n) \leq c\, n \log n \text{ for all } n \geq m_0.$$

We set $m_0 \geq 5$. We prove this by induction.

Base Case: When $n = 5$,

$$\begin{aligned} T(5) &= 2\,T\left(\left\lfloor \frac{5}{2} + \sqrt{5} \right\rfloor\right) + 5 \\ &= 2\,T(4) + 5 \\ &= 2(16) + 5 \\ &= 37 \end{aligned}$$

$$T(5) \leq c\,5 \log 5 \text{ for any } c \geq 1.$$

Induction Hypothesis: We assume that this inequality holds for all m such that $m_0 \leq m < n$. Thus, it holds for $m = \left\lfloor \frac{n}{2} + \sqrt{n} \right\rfloor$, ie.

$$T\left(\left\lfloor \frac{n}{2} + \sqrt{n} \right\rfloor\right) \leq c\left(\left\lfloor \frac{n}{2} + \sqrt{n} \right\rfloor\right) \log\left(\left\lfloor \frac{n}{2} + \sqrt{n} \right\rfloor\right) \tag{1}$$

Induction Step: We can substitute 1 into our original recurrence to get

$$\begin{aligned}
T(n) &\leq 2c\left(\left\lfloor\frac{n}{2}+\sqrt{n}\right\rfloor\right)\log\left(\left\lfloor\frac{n}{2}+\sqrt{n}\right\rfloor\right)+n \\
&\leq 2c\left(\frac{n}{2}+\sqrt{n}\right)\log\left(\frac{n}{2}+\sqrt{n}\right)+n \qquad (\because \lfloor x\rfloor \leq x) \\
&= (cn+2c\sqrt{n})\log\left(\frac{n}{2}\left(1+\frac{2}{\sqrt{n}}\right)\right)+n \\
&= cn\log\left(\frac{n}{2}\left(1+\frac{2}{\sqrt{n}}\right)\right)+2c\sqrt{n}\log\left(\frac{n}{2}\left(1+\frac{2}{\sqrt{n}}\right)\right)+n \\
&= cn\log(n/2)+cn\log\left(1+\frac{2}{\sqrt{n}}\right)+2c\sqrt{n}\log(n/2)+2c\sqrt{n}\log\left(1+\frac{2}{\sqrt{n}}\right)+n \\
&= cn\log(n)-cn\log 2+(cn+2c\sqrt{n})\log\left(1+\frac{2}{\sqrt{n}}\right)+2c\sqrt{n}\log(n)-2c\sqrt{n}\log 2+n \\
&= cn\log(n)-cn+(cn+2c\sqrt{n})\log\left(1+\frac{2}{\sqrt{n}}\right)+2c\sqrt{n}\log(n)-2c\sqrt{n}+n \\
&\leq cn\log(n)-cn+(cn+2c\sqrt{n})\frac{2}{\sqrt{n}}+2c\sqrt{n}\log(n)-2c\sqrt{n}+n \qquad (\because \log(1+x)\leq x) \\
&= cn\log(n)-cn+cn\frac{2}{\sqrt{n}}+2c\sqrt{n}\frac{2}{\sqrt{n}}+2c\sqrt{n}\log(n)-2c\sqrt{n}+n \\
&= cn\log(n)-cn+\cancel{2c\sqrt{n}}+4c+2c\sqrt{n}\log(n)-\cancel{2c\sqrt{n}}+n \\
&= cn\log(n)+4c-cn+n+2c\sqrt{n}\log(n)
\end{aligned}$$

We can choose suitable values of c and m_0 for this inequality to hold. For example, for $c = 1000$ and for all $n \geq 260$,

$$\begin{aligned}
T(n) &\leq cn\log(n)+4c-cn+n+2c\sqrt{n}\log(n) \\
&\leq cn\log(n) \qquad (\because (4c-cn+n+2c\sqrt{n}\log(n)) < 0 \text{ for } c = 1000 \text{ and } n > 260)
\end{aligned}$$

$$\therefore T(n) = O(n\log n)$$

□

2(b)

We specify the function as:

$$f(n) = \frac{-n}{2}+\frac{1}{100}$$

3(a)

The degree of the remainder in polynomial division is at most 1 less than the degree of the divisor. Since $(x - z)$ is a polynomial divisor of degree 1, hence the degree of the remainder is at most 0. This means that the remainder is a constant. Let us denote this constant by k.

We know that

$$A(x) = (x - z)q(x) + k,$$

where $q(x)$ is the quotient that we get on dividing $A(x)$ by $(x-z)$, and $k = A(x) \bmod (x-z)$. Substituting z in this equation, we get

$$\begin{aligned} A(z) &= (z - z)q(z) + k \\ \implies A(z) &= k \end{aligned}$$

Thus, $A(x) \bmod (x - z) = A(z)$.

3(b)

Part 1

From the definition in the question, $Q_{kk}(x) = A(x) \bmod P_{kk}(x)$ where $P_{kk}(x) = (x - x_k)$. As in part (a), since $(x - x_k)$ is a polynomial divisor of degree 1, Q_{kk} has to be a constant since its degree is both at most and at least $k - k = 0$. Thus,

$$\begin{aligned} A(x) &= q(x) \cdot P_{kk}(x) + Q_{kk}(x), \qquad \text{where } q(x) \text{ is the quotient.} \\ A(x_k) &= q(x_k) \cdot (x_k - x_k) + Q_{kk}(x) \\ \implies Q_{kk}(x) &= A(x_k) \end{aligned}$$

Part 2

$$\begin{aligned} &Q_{0,n-1}(x) = A(x) \bmod P_{0,n-1}(x) \\ &P_{0,n-1}(x) = (x - x_0)(x - x_1)(x - x_2) \dots (x - x_{n-1}) \\ &P_{0,n-1}(x) \text{ is a polynomial of degree } n. \\ &A(x) \text{ is a polynomial of degree } n - 1. \end{aligned}$$

$$A(x) = q(x) \cdot P_{0,n-1}(x) + Q_{0,n-1}(x)$$

q(x) has to be 0.
Thus, the quotient is 0, and the remainder is $A(x)$ itself.
$A(x) = Q_{0,n-1}(x)$.

3(c)

Part 1

$$A(x) = q_1(x)\, P_{ij}(x) + Q_{ij}(x) \tag{2}$$

$$Q_{ij}(x) = q_2(x)\, P_{ik}(x) + Q_{ij}(x) \bmod P_{ik}(x) \tag{3}$$

Substituting (3) in (2), we get

$$A(x) = q_1(x)\, P_{ij}(x) + q_2(x)\, P_{ik}(x) + Q_{ij}(x) \bmod P_{ik}(x) \tag{4}$$

We can write $P_{ij}(x)$ as $P_{ik}(x)\, P_{k+1,j}(x)$. Thus, we can rewrite (4) as

$$A(x) = (q_1(x)\, P_{k+1,j}(x) + q_2(x))\, P_{ik}(x) + Q_{ij}(x) \bmod P_{ik}(x) \tag{5}$$

which can be rewritten as

$$A(x) = q_3(x)\, P_{ik}(x) + r(x), \tag{6}$$

where $r(x) = Q_{ij}(x) \bmod P_{ik}(x)$. We know by definition that $r(x) = Q_{ik}(x)$.

Thus, $Q_{ik}(x) = Q_{ij}(x) \bmod P_{ik}(x)$.

Part 2

$$A(x) = q_1(x)\, P_{ij}(x) + Q_{ij}(x) \tag{7}$$

$$Q_{ij}(x) = q_2(x)\, P_{kj}(x) + Q_{ij}(x) \bmod P_{kj}(x) \tag{8}$$

Substituting (8) in (7), we get

$$A(x) = q_1(x)\, P_{ij}(x) + q_2(x)\, P_{kj}(x) + Q_{ij}(x) \bmod P_{kj}(x) \tag{9}$$

We can write $P_{ij}(x)$ as $P_{i,k-1}(x)\, P_{kj}(x)$. Thus, we can rewrite (9) as

$$A(x) = (q_1(x)\, P_{i,k-1}(x) + q_2(x))\, P_{kj}(x) + Q_{ij}(x) \bmod P_{kj}(x) \tag{10}$$

which can be rewritten as

$$A(x) = q_3(x)\, P_{kj}(x) + r(x), \tag{11}$$

where $r(x) = Q_{ij}(x) \bmod P_{kj}(x)$. We know by definition that $r(x) = Q_{kj}(x)$.

Thus, $Q_{kj}(x) = Q_{ij}(x) \bmod P_{kj}(x)$.

3(d)

From part 3(b),

$$A(x) = Q_{0,n-1}(x)$$

and from part 3(c),

$$Q_{0,n/2-1}(x) = Q_{0,n-1}(x) \bmod P_{0,n/2-1}(x)$$

$$Q_{n/2,n-1}(x) = Q_{0,n-1}(x) \bmod P_{n/2,n-1}(x)$$

Thus, to compute $A(x)$, we can recursively split $Q_{0,n-1}(x)$ into 2 equal parts until we reach $Q_{ii}(x)$. From part 3(b), we know that $Q_{ii}(x) = A(x_i)$, and so we can find the values of $A(x_0), A(x_1), \cdots, A(x_{n-1})$ from the corresponding values of the remainders $Q_{00}(x), Q_{11}(x), \cdots, Q_{n-1,n-1}(x)$. Since it is given that computing a remainder takes $O(n \lg n)$ time, this recurrence takes the form $T(n) = 2T(n/2) + O(n \lg n)$. We proved in class that the time complexity of this recurrence is $O(n \lg^2 n)$.

4(a)

We arrange the tasks in increasing order of deadline, and then number them from 1 to n. We fill up a table S whose rows are indexed from 0 to n where row i denotes that we have considered tasks 0 to i, and whose columns are numbered from 0 to $\sum_n^{i=1} e_i$, where e_i is the execution time of task i in the sorted order. Element S_{ij} of the table denotes the value of a schedule after execution happens till time j and we have considered tasks 1 to i. The algorithm works as follows:

$$S_{ij} = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ S_{i-1,j} & \text{if } \min(d_i, j) < e_i, \\ \max((S_{i-1,(\min(d_i,j)-e_i)} + v_i), S_{i-1,j}) & \text{otherwise.} \end{cases}$$

If the time for execution or the number of tasks is 0, $S_{ij} = 0$ as shown in the first recurrence condition.

If the execution time of the function is greater than the time we are considering or the deadline of the task, then we cannot add this task to the schedule and $S_{ij} = S_{i-1,j}$, as shown in the second condition of the recurrence.

Otherwise, this is similar to the knapsack problem and we decide to add the task to the schedule if it contributes to an optimal value and fits in the schedule as shown in the third condition of the recurrence. The final answer is the bottom right entry of the table.

The time taken to sort the tasks is $O(n \log n)$. The number of rows is n and the number of columns $= \sum_n^{i=1} e_i$. Since the maximum value for each e_i is n^c, the number of columns is $nO(n^c) = O(n^{c+1})$. The time to calculate the value of each entry in the table is $O(1)$, and thus the total time complexity of the algorithm is $O(n \log n) + O(n^{c+1}) = O(n^{c+2})$, which is polynomial time complexity.

4(b)

We arrange the tasks in increasing order of deadline, and then number them from 1 to n. We fill up a table S whose rows are indexed from 0 to n where row i denotes that we have considered tasks 0 to i, and whose columns are numbered from 0 to $\sum_n^{i=1} v_i$, where v_i is the value of task i in the sorted order. Element S_{ij} of the table denotes the minimum execution time for scheduling tasks 1 to i with the value of the schedule being at least j. The algorithm works as follows:

$$S_{ij} = \begin{cases} 0 & \text{if } j = 0, \\ \infty & \text{if } j > 0 \text{ and } i = 0, \\ \min(e_i, S_{i-1,j}) & \text{if } v_i \geq 0, e_i \leq d_i \\ \min(S_{i-1,j}, (e_i + S_{i-1,j-v_i})) & \text{if } v_i < j \text{ and } e_i \leq d_i, \\ S_{i-1,j} & \text{otherwise.} \end{cases}$$

The final answer is the maximum value in the last row which is not ∞. The time taken to sort the tasks is $O(n \log n)$. The number of rows is n and the number of columns $= \sum_n^{i=1} v_i$. Since the maximum value for each v_i is n^c, the number of columns is $nO(n^c) = O(n^{c+1})$. The time to calculate the value of each entry in the table is $O(1)$, and thus the total time

complexity of the algorithm is $O(n \log n) + O(n \cdot n^{c+1}) = O(n^{c+2})$, which is polynomial time complexity.